Process Quality Simulator

someonesdad1@gmail.com 22 Dec 2012

Table of Contents

Introduction

What is it?

The Process Quality Simulator (I'll call it PQS for short) is a python script that simulates a production process. What it simulates is the distribution of the product quality **when measurement uncertainty is a significant fraction of the process standard deviation**.

> Here's an example. Suppose we have a process that makes widgets and suppose the stable production process distribution of the value of a quality parameter is normally-distributed with a mean of 1 and a standard deviation of 0.1. Further suppose the specification of a good part is when this quality parameter is in the interval [0.8, 1.1]. It's straightforward to calculate what fraction of the population is bad. Now, however, assume the measurement of the quality parameter also has an uncertainty standard deviation of 0.1. If you do sampling or 100% inspection of the parts from the production process, some parts will be misclassified. A good part with a "true" value of 1.099 of the quality parameter will be incorrectly classified as bad roughly half of the time. Analogously, a bad part with a "true" value of 1.101 of the quality parameter will be incorrectly classified as a good part about half the time.

From this example, you can see that the uncertainty of measurement can sometimes play a big role: here are the classifications that can occur:

	Part is actually:	
Part measures:	**Good**	**Bad**
Good	☺	☹
Bad	☹	☺

If we're responsible for the behavior of the process, we'd like to know about the parts that get misclassified (shown in the table in pink), as these lead to **good parts getting thrown away** (yield loss) and **the customer getting bad parts** (poor customer quality). Statisticians label these classification errors as type 1 and type 2 errors (which number is used depends on the hypothesis being tested), but we'll call them **producer's risk** and **consumer's risk**, as that's what they are typically called in a manufacturing environment.

The case of negligible measurement error can be handled by just looking up the appropriate

percentage points for the production distribution. But when the measurement uncertainty is a significant fraction of the process standard deviation, calculating the type 1 and type 2 errors can be mathematically more difficult. Monte Carlo simulation becomes a good tool to calculate these numbers -- and that's how the PQS module calculates the numbers used in its report and plots.

Why would I want to use it?

The PQS module is aimed at production engineers and managers who want a simple-to-use tool to help them decide where they should put their efforts to improve the quality of a process. The tool can also help quantify what benefits will be reaped if the improvements are successful.

The core code was written to be clear and well-documented, so it should be straightforward to modify things if you want to change the simulation. More importantly, it should be simple enough for a technically-inclined decision-maker to use the simulation and understand how it works by reading the code. A problem with computer models is it can be difficult to convince the decision makers that they should believe the results coming from the models.

What do I need to use it?

Besides the files in the `pqs.zip` package, you'll need to download and install some freely-available tools:

python	A general-purpose programming language. See http://www.python.org/.
numpy	An array processing library for python. See http://www.scipy.org/.

Background

In the early 1980's, I worked in R&D in a division of a large US corporation producing thin film electronic devices. We had a significant investment in engineers who designed and built test equipment to test the devices we made because such tools couldn't be purchased. One of the characteristics of producing these high-tech thin film devices was that a number of the quality parameters had non-trivial measurement uncertainties associated with them.

At the time, I was concerned about the reliability and quality that our envisioned production processes would produce and tried to predict them. While I was able to write down the formal equations for the composite quality, they were analytically intractable, so I resorted to a Monte Carlo simulation to predict the required results (the computers available to us at the time were somewhere between 10 and 100 times slower than the PC I have on my desk today, so these simulations took substantially more time and programming work than they do today). At the time, I had other responsibilities and didn't have time to work on this process quality estimation. However, because I was interested in the problem and its answer, I worked on the problem at home and generated a software package to make the calculations and report the results. The company I worked for gave me permission to produce the software as a product to sell on my own if I wished. A friend and I packaged it up and made a small effort to sell it, but we were a decade or two ahead of the time when people started to focus on quality more, so nothing became of it (besides, both of us had full-time jobs doing other things).

I felt it was time to rewrite this simulator in python and make it available to anyone who wants it. If you're a programmer, take a look at the core code (in `manufacturing.py`), as it's pretty simple -- and you could modify things pretty easily to handle more sophisticated simulations (however, read the next section before making changes).

An observation

The simulation handles one quality parameter. While some people would think this is inadequate for a modern high-tech production process, consider the following observations:

1. The Pareto principle (the vital few and trivial many) virtually always applies to production processes: your yield or quality problem that your management or customer wants fixed is due

to a problem with only one quality parameter.

2. Accurately modeling a complete process would require knowing the distributions of the quality parameters and their uncertainties. More importantly, you have to know the **joint** distributions to predict things accurately, as things are often correlated. In a complex process with lots of people and machines, the effort to know these distributions can be excessive (and the information can be out-of-date rather quickly). A wise production manager wouldn't let such an effort be undertaken unless he/she could be convinced there would be a significant return on investment.

Thus, I think PQS is a suitable tool for almost any production process, at least to give you a high-level understanding of your process capability and to quantitatively answer questions like "What would be the impact of increasing the part specification width by 30%?".

Here's another thing to consider if you're trying to convince people to take (or not take) some action based on some modeling. One of the traps a modeling beginner can fall into is to make the assumption that every detail of the process needs to be included in the model. This can lead to long development times and complicated code and, most importantly, makes it harder to validate the model. Always start with a simple model and see what insight you can get with it.

For an experienced manager or engineer to want to use a model, probably their first requirement of the model is that **it must correctly model situations they know the answer to**. If the model falls on its face at this point, it's doomed. And once you've lost credibility, it's terribly hard to gain it back. On the other hand, if you can get the right answers to simple situations and the model leads to new insights or understanding, it will be valued and used. I believe the `pqs.py` script is a simple enough tool for someone to understand how it works, it's easy to verify known situations, and it will lead to quantitative insights in managing a real-world process. I've used this modeling technique myself when I was a process engineer in a high-tech fab similar to a semiconductor fab.

I wrote the code with lots of comments in it. This was to help the person using the tool understand what is going on in the code. Hopefully, the `pqs.py` script is simple enough that you could even give it to one of the decision-makers and let them play with it and understand it. I know from first-hand experience both as an engineer and manager that decision makers will not believe a computer model or simulation unless they've seen and/or done some significant validation work. In fact, I'd say the validation work is the most important aspect of model development -- if it's not validated, it won't be used -- and then the resources used to develop the model were wasted (nobody wants that on their conscious).

Installation

The `pqs.zip` file from http://code.google.com/p/hobbyutil/ contains the following files:

`manufacture.py`	The python module that does the core part of the simulation.
`pqs.py`	The python script used to run the simulation.
`ProcessSimulator.pdf`	The documentation file you're reading.
`sig.py`	Module used for output of numbers using significant figures.

You'll need to download some freely-available tools:

python	A general-purpose programming language. See http://www.python.org/.
numpy	An array processing library for python. See http://www.scipy.org/.

Install the needed python programming language and libraries as instructed by the installation programs. I recommend that you don't use any space characters in the path to python and the libraries (this can create headaches on some systems).

Then copy the python scripts to a convenient directory and run `pqs.py` in a console window by typing

```
python pqs.py
```

You should see the usage statement for the program. If you get this far, then you can run the examples in the following sections.

Examples

Simplest example

When you run the `pqs.py` script, you need to give it a file on the command line that contains the process and financial components of the model. If you're starting from scratch, run the command

```
python pqs.py -c
```

and a template of this file is printed to stdout. Save it to a file, modify the settings, and run the simulation.

The simulation runs by "manufacturing" parts in lots of a specified size. This was done with the anticipation that a user might want to modify the 100% inspection policy to using sampling inspection instead. If you're interested in getting results quickly with 100% inspection, set the `NumberOfLots` to 1 and the `PartsPerLot` to the total number of parts you want to produce. On my older computer and using normal distributions, the simulation can make over a million parts and complete the report in under a second[1]. If I change to 1000 lots of 1000 parts each, the simulation run time goes up by nearly an order of magnitude.

If you run the command `python pqs.py -c` and put the results into a file `data_file`, the file `data_file` will have the contents

```
# This is a template of the initialization file needed for the use of
# the pqs.py process simulation script.  The lines in this file are
# valid python statements and will be run by the simulation script to
# define the simulation's variables.  This file can only have blank
# lines, comments, and assignment statements.

# Process variables

    # The script is set up to use numpy's normal distribution for both
    # the process distribution and the measurement uncertainty
    # distribution.  If you want to use other distributions, you'll
    # need to use or create a function that takes a mean, standard
    # deviation, and number of parts to make and returns a numpy array
    # of the generated random numbers.
    ProcessDistribution = normal
    ProcessDistributionName = "normal"

    # Process mean
    ProcessMu = 0

    # Process standard deviation
    ProcessSigma = 1

    # The measurement uncertainty associated with each measurement is
    # also a random variable.
    MeasurementDistribution = normal
    MeasurementDistributionName = "normal"

    # Measurement uncertainty distribution mean; note this is really a
    # measurement bias (and is so indicated in the report)
    MeasurementMu = 0
```

1 This is because most of the work is done by numpy in compiled C code.

```
    # Measurement uncertainty distribution standard deviation
    MeasurementSigma = 1e-10

    # For speed, make 1 lot and put all the production parts needed
    # into that lot.  The only reason to do otherwise is if you want
    # to change the manufacturing.py script to do sampling inspection.
    NumberOfLots = 1
    PartsPerLot = 100000

    # These two numbers determine the interval within which a good
    # part lies.  Note:  these default values should produce a
    # report that has no misclassified parts (i.e., no producer's or
    # consumer risk) and a 10% yield loss (-1.64485 is the z-score for
    # 0.05).
    Specification = (-1.64485, 1.64485)

# Financial information.  Note that the simulation and the report
# printed to stdout don't make any assumptions about the monetary
# units.

    # The following cost is the direct manufacturing cost of a single
    # part.  It includes e.g. the raw material, manufacturing cost,
    # and direct labor.
    CostToProducePart = 1

    # The following costs are incurred during the stated action.
    CostToTestPart = 0
    CostToShipPart = 0
    CostToScrapPart = 0

    # The following is a variable overhead cost associated with each
    # part.  The total overhead cost in this category is calculated by
    # multiplying by the number of parts produced.
    CostVariableOverhead = 0

    # The fixed overhead is an amount of money that is needed to make
    # the production run, regardless of the number of parts made.
    # Things that are lumped in this category can be things like
    # floorspace, depreciation, engineering/management salaries,
    # corporate charges, etc.
    CostFixedOverhead = 0

    # The selling price is what the part is sold to the customer for.
    SellingPricePerPart = 2

    # When a bad part is shipped (i.e., a part with its quality
    # parameter outside the specification limits), the part is assumed
    # to fail in the customer's environment and cause the customer to
    # incur the following cost.
    CustomerCostPerBadPart = 0

# General items
    # The Title is a string that will be displayed centered at the
    # beginning of the report.
    Title = ""

    # You can control the number of significant figures that the
    # process characteristics, yields, etc. are printed with.
    SignificantDigits = 3
```

If you then run the command `python pqs.py -s 0 data_file`, you'll get the following report to stdout (the leading line of the date has been chopped off and the lines have been numbered):

```
 1                                  Seed = 0
 2
 3 Source files and their hashes
 4 ----------------------------
 5   aa801b541c4da31ce433ca89ddd7df9f D:/p/math/ProcessAnalyzer/manufacture.py
```

```
  6   1f53fb07fe3cd47593b039a29e912b1b D:/p/math/ProcessAnalyzer/pqs.py
  7
  8 Process characteristics                    (3 significant figures)
  9 ----------------------
 10   Distributions
 11     Process     = normal
 12     Measurement = normal
 13   Process mean                                          0.00
 14   Process standard deviation                            1.00
 15   Measurement bias                                      0.00
 16   Measurement standard deviation                     1.00e-10
 17   Part acceptance interval                     [-1.64, 1.64]
 18   Number of lots made                                      1
 19   Parts per lot                                      100,000
 20
 21 True (unknowable) process output
 22 --------------------------------
 23   Total parts made          100,000
 24   Actual good                90,009    (90.0% of total)
 25   Actual bad                  9,991    (9.99% of total)
 26
 27 What the testing process determined
 28 -----------------------------------
 29   Good tested good           90,009  90.0%
 30   Good tested bad                 0  0.00%  <-- Producer's risk
 31   Bad  tested good                0  0.00%  <-- Consumer's risk
 32   Bad  tested bad             9,991  9.99%
 33   Process mean              0.00682
 34   Process std dev             0.999
 35   Min, max measured    [-4.40, 4.66]
 36
 37 Production results
 38 ------------------
 39   Parts shipped              90,009  0.00% of these are bad parts
 40   Parts scrapped              9,991  0.00% of these are good parts
 41   Yield              actual = 90.0%, measured = 90.0%
 42   Scrapped           actual = 9.99%, measured = 9.99%
 43
 44 Money                                 Amount    Cost per part
 45 -----                               ---------    -------------
 46   Production cost                     100,000        1.000
 47   Testing cost                              0        0.000
 48   Shipment cost                             0        0.000
 49   Cost to scrap                             0        0.000
 50   Variable overhead cost                    0        0.000
 51   Fixed overhead cost                       0        0.000
 52                                       ---------     --------
 53   Total production cost               100,000        1.000 (per part started)
 54                                                      1.111 (per part shipped)
 55   Revenue                             180,018        2.000 selling price
 56   Gross profit                         80,018
 57   % gross profit                          44.4%
 58   Customer's loss                           0        0.000 (per received part)
```

Line 1 gives the seed used for this simulation. The seed is the string used with the `-s` option on the command line (in this case it was `0`). Using the `-s` option allows you to generate the same stream of random numbers in the simulation; thus, if you use the same `data_file`, you should get the same numbers in the output. A title and time/date lines were deleted from the output prior to the seed line, as they will differ from what you see.

Lines 3-6 give MD5 hashes of the two files that make up the script. This output is present to document which version of the scripts were used to generate the output. Of course, you can't identify the files from the hash, so you would be wise to keep the different revisions in a revision control system. Comments at the end of this document indicate how I would change things in a real production environment to know which revision the simulation was run with.

Lines 8-19 give the process characteristics. For this example, the acceptance interval is chosen to be `[-z, z]` where z is a normal deviate such that the area under the normal curve between these two limits is 0.90. The measurement standard deviation is set to a negligibly small number (the numpy `normal` function won't accept an argument of zero).

Lines 21-25 give the fraction of parts produced that were good and bad. These are the "true" values from the process distribution before the measurement uncertainty random number is added in (and are unknowable in principle unless the measurement uncertainty is a tiny fraction of the process standard deviation). You can see that the process' true yield is indeed 90%. These lines tell you about the absolute best behavior you could get from this manufacturing process. Including the measurement of the quality parameter and the subsequent classification of good and bad parts is only going to make the output worse (i.e., increase the number of good parts you throw away and bad parts that you unknowingly ship to your customer).

Lines 27-35 show how the measurement has classified the parts. Because there is no measurement uncertainty in this example, consumer's and producer's risks are zero. The measured process mean and standard deviation are close to the true values as set in the `data_file`.

Lines 37-42 show what parts are shipped and scrapped. If there was concommitant measurement uncertainty, the measured yield and scrap values would differ from the actual yield and scrap values.

Lines 44-58 show the monetary aspects of the production process. In line 55, revenue is the cost per part multiplied by the number of parts shipped. Gross profit is the total production cost in line 53 subtracted from the revenue and the % gross profit is the gross profit as a percentage of the revenue.

The customer's loss in line 58 is the customer's cost of poor quality. This number is the `data_file`'s `CustomerCostPerBadPart` number multiplied by the number of parts shipped times the bad part percentage given on line 39.

If you examine the `manufacture.py` script, you'll see that only six different numbers need to be accumulated. The first two are given in lines 24-25 and are the true number of good and bad parts made. The remaining four numbers accumulated are lines 29-32.

A more typical example

Using the following `data_file`

```
# Process
    ProcessDistribution = normal
    ProcessDistributionName = "normal"
    ProcessMu = 10
    ProcessSigma = 1
    MeasurementDistribution = normal
    MeasurementDistributionName = "normal"
    MeasurementMu = 0
    MeasurementSigma = 0.5
    NumberOfLots = 1
    PartsPerLot = 423110
    Specification = (9.0, 11.0)

# Financial
    CostToProducePart = 8.43
    CostToTestPart = 0.92
    CostToShipPart = 0.3
    CostToScrapPart = 0.55
    CostVariableOverhead = 0.28
    CostFixedOverhead = 140000
    SellingPricePerPart = 33.5
    CustomerCostPerBadPart = 35
```

the following report was gotten (seed was 0 and hashes are eliminated):

```
Process characteristics                    (3 significant figures)
```

```
-----------------------
  Distributions
    Process     = normal
    Measurement = normal
  Process mean                                        10.0
  Process standard deviation                          1.00
  Measurement bias                                    0.00
  Measurement standard deviation                     0.500
  Part acceptance interval                  [9.00, 11.0]
  Number of lots made                                    1
  Parts per lot                                   423,110

True (unknowable) process output
--------------------------------
  Total parts made          423,110
  Actual good               288,632    (68.2% of total)
  Actual bad                134,478    (31.8% of total)

What 100% inspection determined
-------------------------------
  Good tested good          236,781  56.0%
  Good tested bad            51,851  12.3%  <-- Producer's risk
  Bad  tested good           29,367  6.94%  <-- Consumer's risk
  Bad  tested bad           105,111  24.8%
  Process mean                 10.0
  Process std dev              1.12
  Min, max measured    [4.53, 15.4]

Production results
------------------
  Parts shipped             266,148  18.0% of these are bad parts
  Parts scrapped            156,962  21.8% of these are good parts
  Yield              actual = 68.2%, measured = 62.9%
  Scrapped           actual = 31.8%, measured = 37.1%

Money                              Amount   %   Cost per part
-----                            --------- ----- -------------
  Production cost                3,566,817  78.5    8.430
  Testing cost                     389,261   8.6    0.920
  Shipment cost                    244,856   5.4    0.300
  Cost to scrap                     86,329   1.9    0.550
  Variable overhead cost           118,470   2.6    0.280
  Fixed overhead cost              140,000   3.1    0.331
                                 --------- ----- --------
  Total production cost          4,545,734 100.0   10.744 per part started
                                             17.080 per part shipped
  Revenue                        8,915,958         33.500 selling price
  Gross profit                   4,370,223
  % gross profit                      49.0%
  Customer's loss                1,027,845          3.862 per received part
```

Suppose this run simulates one month's production. An experienced process person would of course see the poor process capability by looking at the specification and process parameters. However, I've been in situations where the business requirements dictated that we had to live with such a process for a while (a new production line coming up for the first time could have numbers like this). The report lets you see immediately that you have a yield loss of about 1/3, you're throwing 12% of your good parts away and shipping 7% of your bad parts to your customer. The process definitely needs attention. While the revenue and profit might look acceptable, the large financial loss to the customer due to poor product quality won't be a situation anyone wants to continue. Or at least the decision makers will recognize an opportunity for improvement. ☺

Final comments

For me, a key in the validation of a model is that it produces correct answers to problems where I

know what the answer should be. Thus, for the `pqs.py` script, setting the measurement bias and standard deviation to zero and a very small number, respectively, and having the process distribution be a normal distribution will give you results that can be calculated using a table of the normal distribution's cumulative distribution function. A first validation step can be checking a few such cases.

Checking the output of the simulation when measurement uncertainty is not zero is harder. It would be possible to set up a numerical integration to perform the calculation, but I've never bothered because a Monte Carlo simulation is much easier to get working. Further checking of this case can be done by making small increases in the measurement standard deviation and seeing if the increases in the producer's and consumer's risks seem appropriate.

The script shouldn't be too hard to modify to allow for sampling inspection if you wish. I've never bothered with this because when I needed the tool, our factory was doing 100% inspection. And I know that sampling inspection will just increase the producer's and consumer's risk over and above the 100% inspection case.

It's also likely that you'd want to change the script to reflect your own process nomenclature and financial reporting numbers. This is fine, but a practical warning is to change the code that does the reporting one item at a time, as when there's a mismatch in a formatting code somewhere, it can be hard to track down (this is a weakness in python's string formatting mini-language).

Once you get the scripts to a state where you want to use them for "production" decision-making, I recommend you make sure they're tracked in a revision control system and that you can get back to older versions when needed. There are many different revision control tools to choose from (both open source and commercial tools); if you want an open source distributed tool (a popular modern architecture), take a look at git, Mercurial, or Bazaar.

> Aside: if I was doing production work with this model, I'd keep it in a revision control system and add some code into the script that would ensure the simulation being run was from scripts that were in the revision control system; I would put their revision numbers into the report instead of the hashes. This is pretty easy to do with the python `subprocess` module, but I didn't put it into the script because the details depend on what revision control system you use. One of the advantages of such checking is that you can then ensure that a simulation is only run with scripts that are checked-in (and thus repeatable). A similar thing could be done for the `data_file`.

If I was still working in a production environment, I'd enjoy having access to a tool like the `pqs.py` script. I'd like to see the management and engineering staff use the tool to help understand their processes -- and I'd even encourage the production operators and maintenance techs to do so too.

If you do decide to build a more complex model for your production process, consider making this more complex model operate in a mode where the individual contributing process' outputs can be predicted by the `pqs.py` script. This could then be an important validation step of the more complex model.

One aspect that the `pqs.py` model doesn't simulate is a group of final test machines. For example, in the factory I worked in, the final test machines numbered between 5 and 10 (I've forgotten the exact number) and they were relatively complex beasts with robot part handling and significant maintenance effort to be brought on-line and kept there (and this was also in a highly-filtered clean room, adding to the complexity of keeping things running). The process output was a mixture of the characteristics of the test machines (and there were also multiples of the large pieces of capital equipment that were needed by the previous manufacturing steps). Thus, the real output of the process would be a mixture of a number of separate processes. While it would be tempting to construct a model of the factory that reflected this additional complexity, I'd suggest keeping things simple and e.g. just adding in some measurement bias to reflect the influence of different machines. You could combine the results manually and get an idea of the overall effects. I've constructed a lot of computer models and virtually always find they take much more effort to develop and validate

than you think they will (and convincing a decision-maker that the model makes sense can also take a lot of work). Even if you do decide to invest in the development of a more complex model, this preliminary piecemeal checking can provide good validation test cases for the more complex model, as long as you keep good records and make good use of revision control.

One of the aspects of a complex production process with many machines and parallel paths through production is that the composite output can have a multi-modal and truncated distribution. The power of a Monte Carlo simulation is that these situations can be modeled almost as easily as the simple situations, even though the problem would be nearly analytically intractable. For a hint of how such a thing might be done, see the next section.

Other distributions

If you want to use other distributions for either the manufacturing or measurement processes, first check what support is supplied by numpy (numpy supports many other discrete and continuous distributions). If you don't find what you need, consider downloading scipy, as it supports even more distributions. Finally, you can write your own function. Here's an example of a function that produces a truncated normal distribution where the values returned must be greater than zero:

```
import numpy as np
from numpy.random import normal

def normal_trunc(mean, standard_deviation, sample_size):
    assert mean >= 0 and sample_size > 0
    x = np.arange(0).astype(float)
    while len(x) < sample_size:
        x = np.concatenate((x, normal(mean,  standard_deviation, sample_size)))
        x = np.compress(x > 0, x)
    return x[:sample_size]
```

The `assert` line is important to ensure that the function doesn't create an infinite loop; if the mean was negative and the standard deviation was small, the random number generator might never generate any positive numbers.

You'd insert this function into the global namespace of the `pqs.py` script; the easiest way is to import it from another file. Suppose the function above is in a file called `normaltrunc.py`. Then you'd add the line

```
from normaltrunc import normal_trunc
```

to the top of the `pqs.py` script and you could then use this distribution in your simulation.